Experiment-2

Feedforward Neural Network (MLP)

Aim:

To explore the structure and training of Feedforward Neural Network (MLP) and to visualize forward and backpropagation flow and hidden-layer activations.

Theory:

Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are the essential deep learning models. The goal of a feedforward network is to approximate some function f *. For example, for a classifier, y = f *(x) maps an input x to a category y. A layered feedforward network is one such where any path from an input node to an output node traverses the same number of layers e.g., the n^{th} layer of such a network consists of all nodes which are n arc traversals from an input node. A hidden layer is one which contains hidden nodes. Such a network is fully connected if each node in layer "I" is connected to all nodes in layer (i+1) for all "i". Layered feedforward networks have become very popular as they have been found in practice to generalize well, i.e. when trained on a relatively sparse set of data points, they will often provide the right output for the testing set. When we use a feedforward neural network to accept an input x and produce an output yˆ, information flows forward through the network.
The inputs x provides the initial information that then propagates up to the hidden units at each layer and finally produces yˆ. This is called forward propagation. During training, forward propagation can continue onward until it produces a scalar cost of J(θ). The back-propagation algorithm (Rumelhart et al., 1986a), often simply called backprop, allows the information from the cost to then flow backwards through the network, to compute the gradient. The backpropagation algorithm governs the learning process of the feedforward neural network by computing gradients efficiently, can often find a good set of weights (and biases) in a reasonable amount of time. Backpropagation is a variation in gradient search. It generally uses a least-squares optimal criterion. The key to backpropagation is a method for calculating the gradient of the error with respect to the weights for a given input by propagating error backwards through the network.

Layer-by-Layer (MLP) Transformation:

- **Input Layer:** Receives raw input x.
- **First Hidden Layer:** $h1 = g1(W1 \cdot x + b1)$
- **Second Hidden Layer (if any):** $h2 = g2(W2 \cdot h1 + b2)$
- **Output Layer:** $y = gn(Wn \cdot h(n-1) + bn)$

This can be expressed more generally as a composition of functions:

$$\Phi(x) = Wn \cdot \rho \cdot W(n-1) \cdot \ldots\ldots. \cdot \rho \cdot W1(x)$$

Where:

- $Wi = Ai \cdot x + bi$ represents the linear transformation (weights and biases) for layer i.
- ρ is the activation function (often the same across layers).

Neural networks where the output from one layer is used as input to the next layer. Such networks are called feedforward neural networks. This means there are no loops in the network - information is always fed forward, never fed back.

These models are called feedforward because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are fed back into itself.

I. **Gradient Based Learning:** For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values. The iterative gradient-based optimization algorithms used to train feedforward networks and almost all other deep models.

II. **Learning XOR:** To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function. The XOR function (“exclusive or”) is an operation on two binary values, x1 and x2. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0. The XOR function provides the target function y = f*(x) that we want to learn. Our model provides a function y = f(x;θ) and our learning algorithm will adapt the parameters θ to make f as similar as possible to f*.

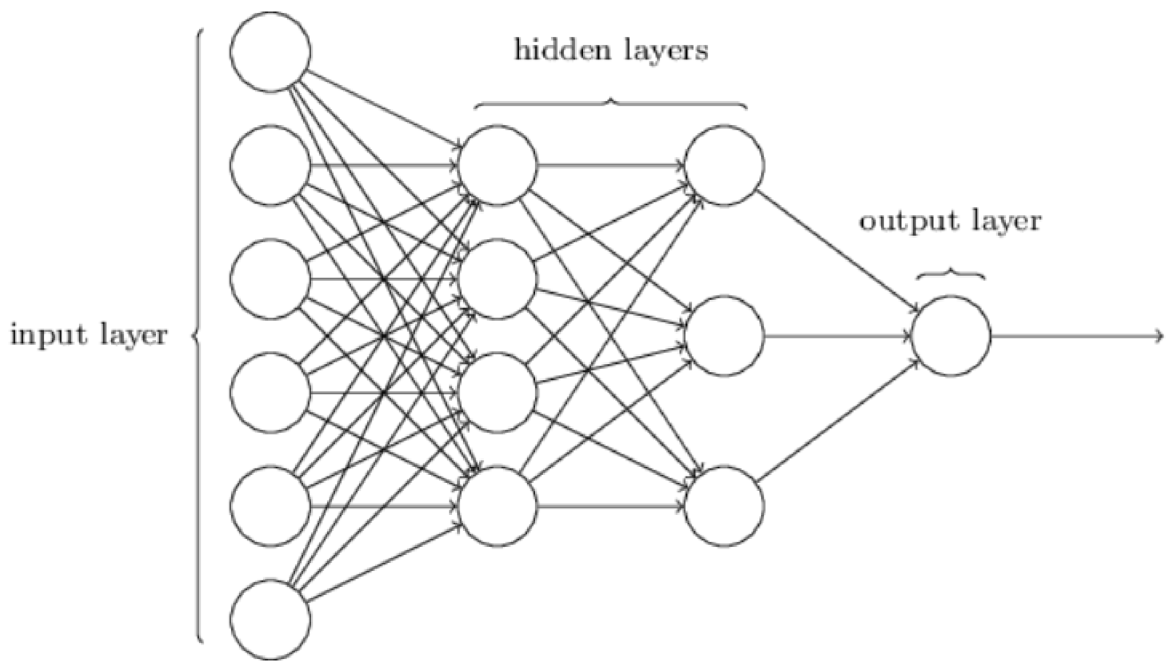


Figure 1- Architecture of Feedforward Neural Network

(Source: M. A. Nielsen, Neural Networks, and Deep Learning.)

The process of forward propagation from input to output and backward propagation of errors is repeated several times until the error gets below a predefined threshold. The whole process is represented in the following diagram:

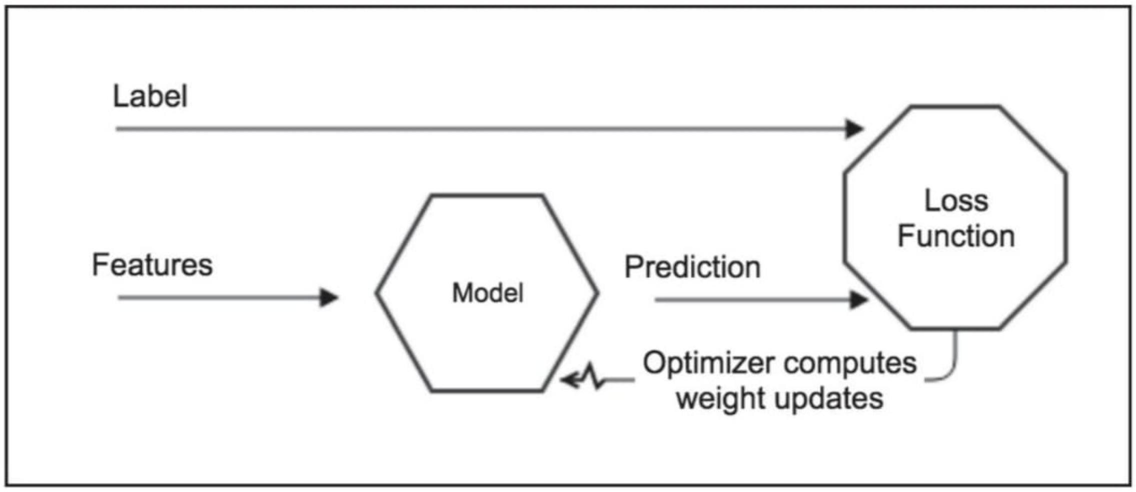


Figure 2- MLP Process both Forward and Backprop

(Source: Antonio Gulli, Sujit Pal, Deep Learning with Keras)

The features represent the input, and the labels are here used to drive the learning process. The model is updated in such a way that the loss function is progressively minimized. In a neural network, what really matters is not the output of a single neuron, but the collective weights adjusted in each layer. Therefore, the network progressively adjusts its internal weights in such a way that the prediction increases the number of labels correctly forecasted. Of course, using the right set features and having quality labeled data is fundamental to minimizing the bias during the learning process.

Merits of Feedforward Neural Network (MLP):

- **Scalability:** The number of hidden layers and neurons can be adjusted to fit the complexity of the problem.
- **Performance on Tabular Data:** In many research contexts, particularly with structured or tabular datasets, MLPs can outperform more complex models in certain tabular data scenarios due to their simplicity and ability to learn direct features.
- **Universal Function Approximation:** MLPs can approximate virtually any continuous function, allowing them to model highly complex, non-linear relationships.

Demerits of Feedforward Neural Network (MLP):

- **Sensitivity to Hyperparameters:** Performance heavily depends on carefully choosing the number of layers, nodes, learning rates, and activation functions.
- **Overfitting:** MLPs are prone to learning training data too well, resulting in poor generalization of unseen data, especially with smaller datasets.
- **Gradient Issues:** Deep networks can encounter vanishing or exploding gradients, making training difficult.
- **Data Requirements:** They often require sufficiently large, labelled datasets for effective training.

Code and Result:

Block 1: Importing Required Libraries
INPUT:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelBinarizer
from sklearn.metrics import classification_report, confusion_matrix

import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.utils import plot_model
from IPython.display import Image, display
print ("Libraries imported")
```

OUTPUT:

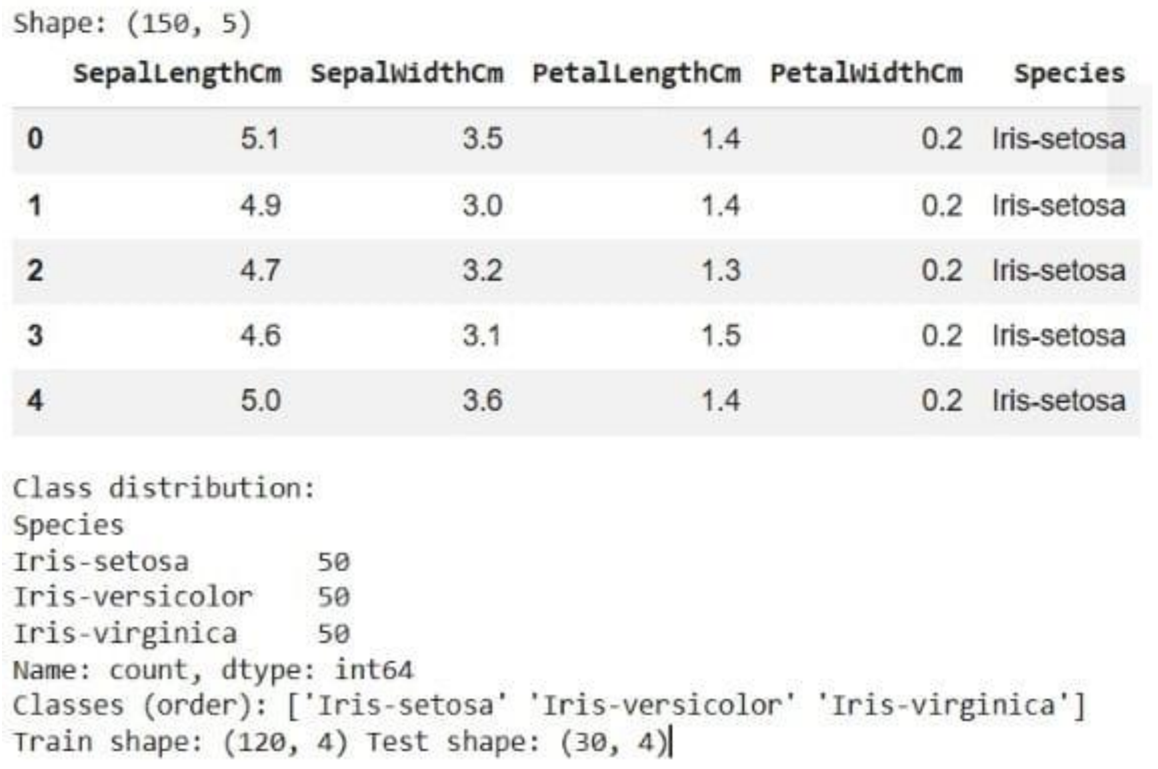
```
Libraries Imported
```

Block 2: Loading Dataset
INPUT:

```
FILE_PATH = "Iris.csv"
data = pd.read_csv(FILE_PATH)
if "Id" in data.columns:
```

```
data = data.drop("Id", axis=1)
print("Shape:", data.shape)
display(data.head())
print(data["Species"].value_counts())
X = data.drop("Species", axis=1). values
y_species = data["Species"]. values
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
binarizer = LabelBinarizer()
y_encoded = binarizer.fit_transform(y_species)
print ("Classes (order):", binarizer.classes_)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.2, random_state=42,
stratify=y_encoded)
print ("Train shape:", X_train.shape, "Test shape:", X_test.shape)
```

OUTPUT:



```
Shape: (150, 5)
```

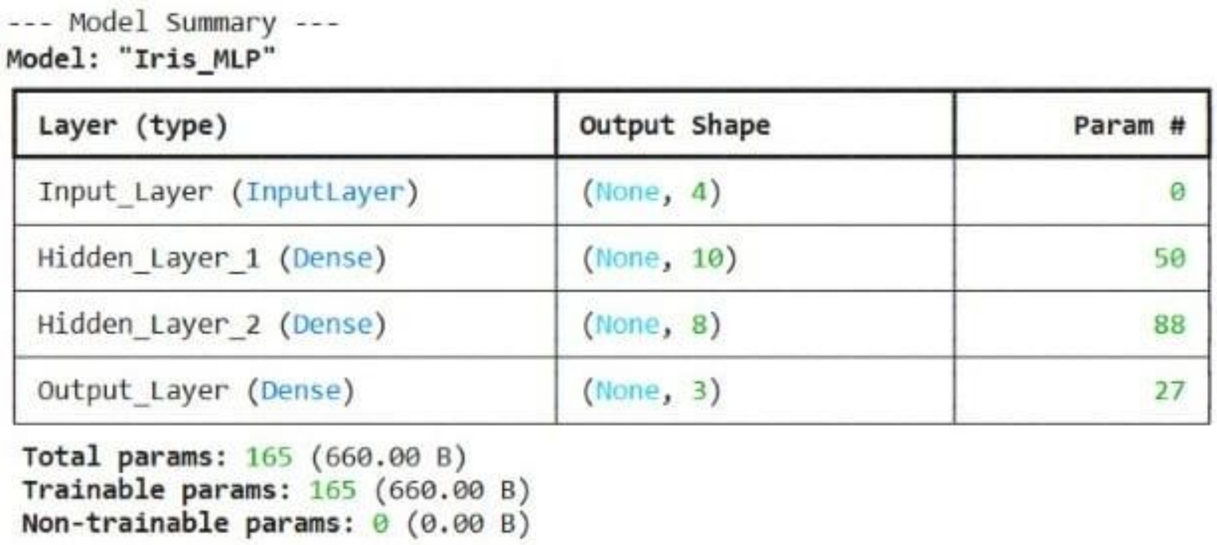
	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
Class distribution:
Species
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: count, dtype: int64
Classes (order): ['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']
Train shape: (120, 4) Test shape: (30, 4)
```

Block 3: Initializing Parameter & Model Building
INPUT:

```
EPOCHS = 100
BATCH_SIZE = 8
LEARNING_RATE = 0.01
OPTIMIZER = "RMSprop"

input_dim = 4
inputs = Input(shape=(input_dim,))
x1 = Dense(10, activation="relu")(inputs)
x2 = Dense(8, activation="relu")(x1)
outputs = Dense(3, activation="softmax")(x2)
model = Model(inputs, outputs)
model.compile(
   optimizer=tf.keras.optimizers.Adam(0.001),
   loss="categorical_crossentropy",
   metrics=["accuracy"]
)
model.summary()
try:
   plot_model(
      model,
      to_file="mlp_model.png",
      show_shapes=True,
      show_layer_names=True,
      dpi=96,
   )
   print ("\n MLP architecture diagram:")
   display (Image(filename="mlp_model.png"))
except Exception as e:
   print ("\nCould not plot model diagram:", e)
```

OUTPUT:

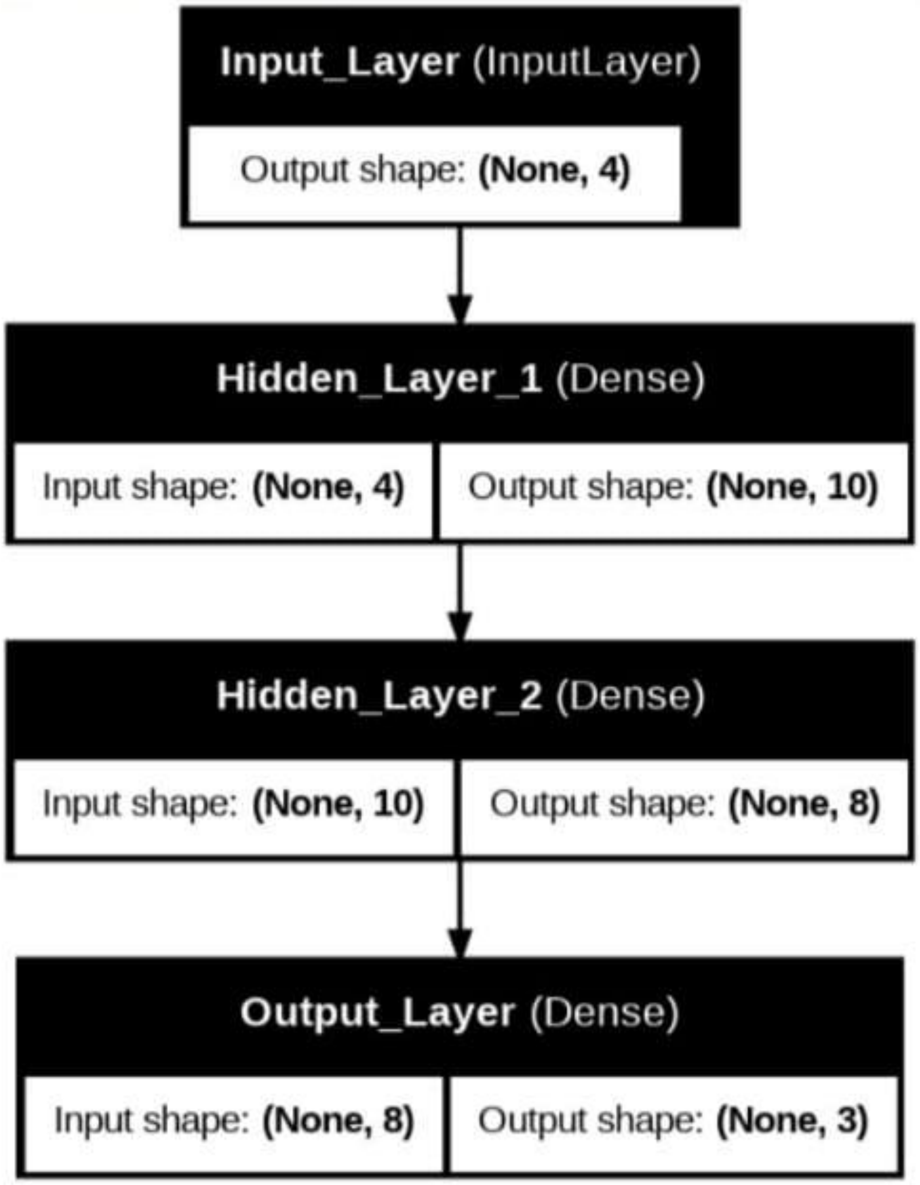


```
--- Model Summary ---
Model: "Iris_MLP"
```

Layer (type)	Output Shape	Param #
Input_Layer (InputLayer)	(None, 4)	0
Hidden_Layer_1 (Dense)	(None, 10)	50
Hidden_Layer_2 (Dense)	(None, 8)	88
Output_Layer (Dense)	(None, 3)	27

```
Total params: 165 (660.00 B)
Trainable params: 165 (660.00 B)
Non-trainable params: 0 (0.00 B)
```

```
MLP architecture diagram:
```

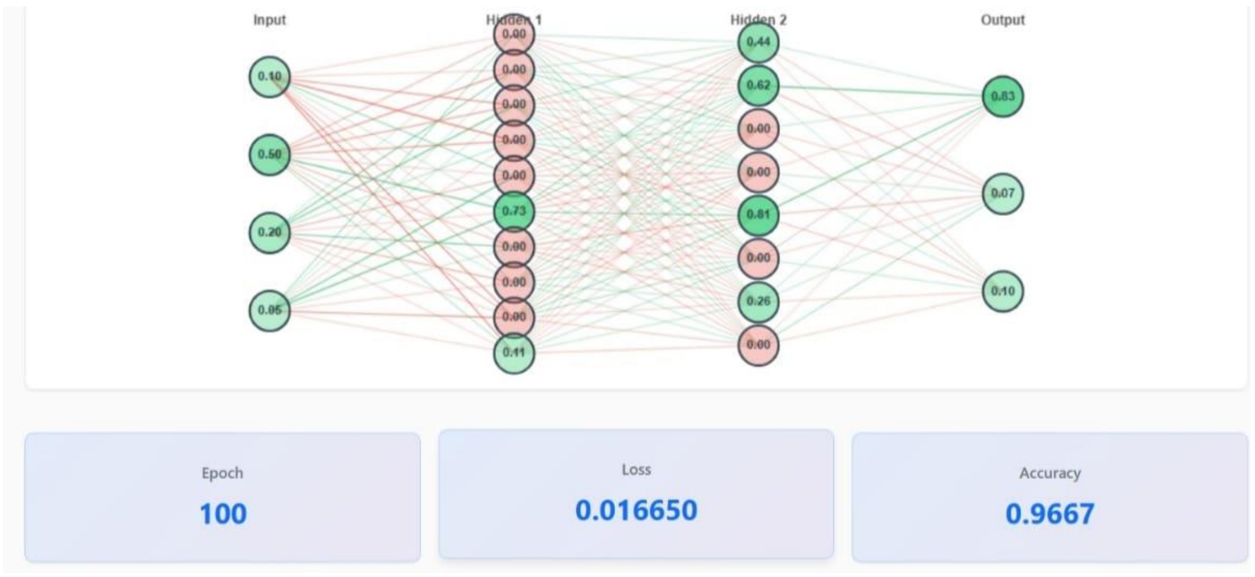


Block 4: Model Training

INPUT:

```
history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=8,
    validation_split=0.1,
    verbose=0
)

model.compile(
    optimizer=tf.keras.optimizers.RMSprop(
        learning_rate=0.01
    ),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

print("Training Completed!")
```

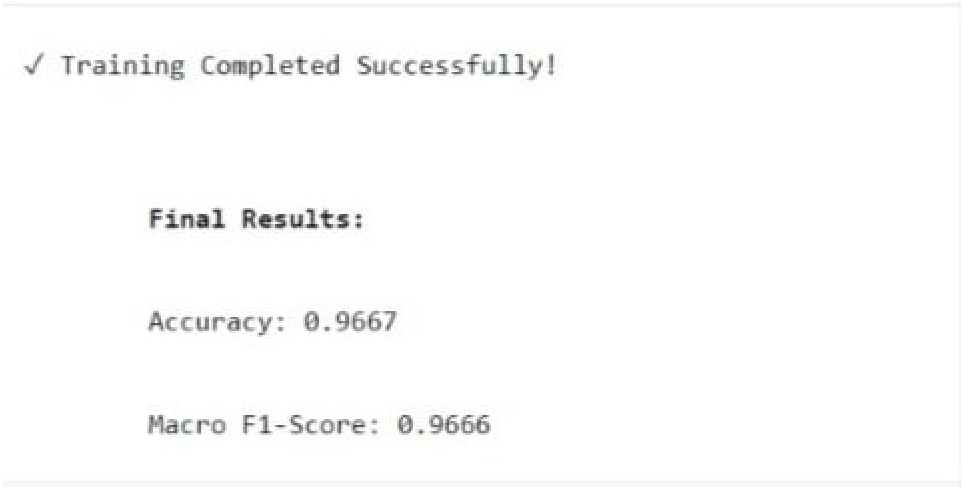
OUTPUT:



Epoch
100

Loss
0.016650

Accuracy
0.9667



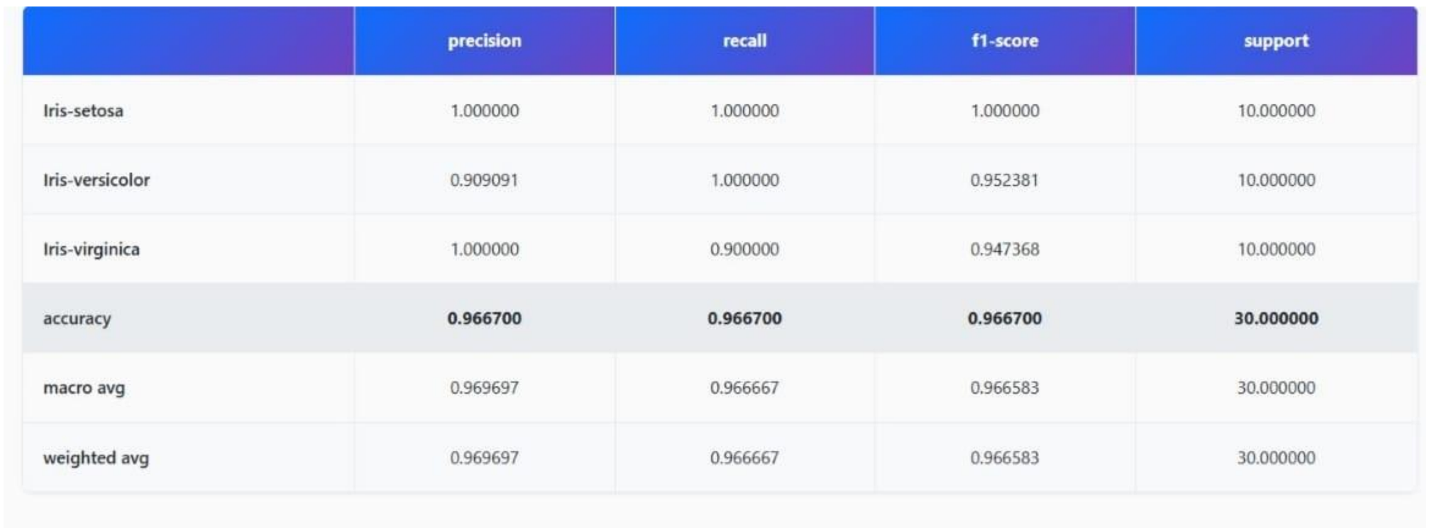
```
✓ Training Completed Successfully!


        Final Results:

        Accuracy: 0.9667

        Macro F1-Score: 0.9666
```

Block 5: Model Evaluation

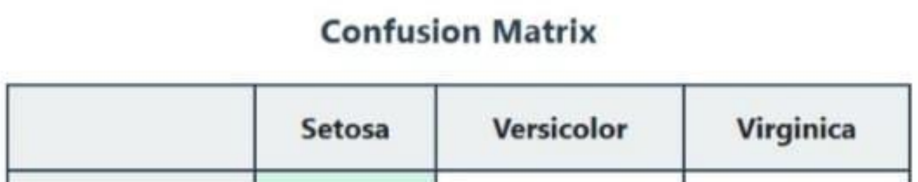
INPUT:

```
y_true = binarizer.inverse_transform(y_test)
y_pred = binarizer.inverse_transform(
   model.predict(X_test, verbose=0)
)

report = pd.DataFrame(
   classification_report(y_true, y_pred, output_dict=True)
).transpose()

display(report)

# Confusion Matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(6,5))
plt.imshow(cm, cmap="Blues")
plt.title("Confusion Matrix")
plt.colorbar()
plt.show()
```

OUTPUT:



	precision	recall	f1-score	support
Iris-setosa	1.000000	1.000000	1.000000	10.000000
Iris-versicolor	0.909091	1.000000	0.952381	10.000000
Iris-virginica	1.000000	0.900000	0.947368	10.000000
accuracy	**0.966700**	**0.966700**	**0.966700**	**30.000000**
macro avg	0.969697	0.966667	0.966583	30.000000
weighted avg	0.969697	0.966667	0.966583	30.000000



Confusion Matrix

	Setosa	Versicolor	Virginica